Controlling the Elasticity of Web Applications on Cloud Computing

Extended Version

Michel Albonico
AtlanMod team (Inria, Mines Nantes, Lina)
UTFPR, Brazil and Mines Nantes, France
michelalbonico@utfpr.edu.br

Jean-Marie Mottu
AtlanMod team (Inria, Mines Nantes, Lina)
University of Nantes
jean-marie.mottu@inria.fr

Gerson Sunyé
AtlanMod team (Inria, Mines Nantes, Lina)
University of Nantes
gerson.sunye@univ-nantes.fr

ABSTRACT

Web applications are often exposed to unpredictable workloads, which makes computing resource management difficult. The resource may be overused when the workload is high and underused when the workload is low. A solution to deal with unpredictable workloads is to migrate Web applications to cloud computing infrastructures, where the resource is varied according to demand, i. e., elasticity. With elasticity, all the resource variations happen during the Web application runtime. To deal with this, the Web application, and its service layers must behave in an elastic manner, which comprises adaptation tasks. These tasks may introduce functional and non-functional errors into the Web application. To find these errors, we must test the Web application when the adaptation tasks are performed, during the resource variations. Some tests may require a specific sequence of resource variations, which are difficult to be achieved without controllability. Therefore, in this paper, we propose an approach that controls the required resource variations. We validated our approach by conducting several experiments on Amazon EC2 cloud infrastructures. In these experiments, we successfully lead the Web application through the required resource variations.

CCS Concepts

•**Networks** → *Cloud computing;* •**Information systems** → *Web applications;* •**Software and its engineering** → *Organizing principles for web applications;*

Keywords

Web application, elasticity, elasticity states, controllability, cloud computing.

1. INTRODUCTION

Web application workloads vary in an unpredictable way. This variation may lead to both, resource overuse or underuse, when the workload is high or low. A way to improve the workload variations processing is by migrating Web applications to cloud computing infrastructures, which provide elasticity, i. e., the resource is varied according to demand.

Elastic infrastructures vary the resource (allocate, and deallocate) at application runtime. To deal with these variations, the Web application and its service layers (database, application container, etc.) must behave in an elastic manner. This comprises adaptation tasks [7], which may be represented by the operations listed by Bersani et al. [3]: component synchronization, registration, and data replication.

Adaptation tasks may introduce functional, and non-functional errors into the Web application. For instance, a functional error may happen due to an inconsistency on data replication, which may result in an unexpected Web page content. Non-functional errors, such as rejected requests and long response times, may result from a delay in a new resource allocation (e. g., Web server), or a problem in reconfiguring the load balancer. To find these errors, the application must be tested when the adaptation tasks are performed. Otherwise, they do not occur, and cannot be discovered.

Testing Web applications through elasticity may be impracticable without controllability. Each test may require specific elastic behavior, which implies in an exact sequence of resource variations. This sequence may take too long to happen, or not happen when it is not controlled. Therefore, it is necessary an approach that ensures the reproducibility of resource variations according to the test needs.

The reproducibility of resource variations may be controlled by exposing the Web application to proper workload variations. Regarding the generation of Web application workload, some work addressed this topic for the last years [14, 13, 10, 8], though none of these work considers elasticity. Other work [9, 5] vary the workload expecting by resource variations. However, they do not control the resource variations.

In this paper, we propose an approach that controls the generation of workload variations according to the required resource variations. We also propose an implementation of this approach, which executes automatically. This implementation is executed in three phases: profiling, workload calculation and application leading. In the profiling phase, the application is exposed to a workload pattern, then we gather the resource usage profile related to this workload pattern. In the workload calculation phase, we estimate (based on the resource usage profile) the workload intensities necessary to request the required resource variations. Then, in the application leading phase, we expose the application to each one of the calculated workload intensities until the resource variations are completed.

We validate our approach conducting experiments on

Amazon EC2[1]. In these experiments, we use the bulletin-board benchmark application RUBBoS[2] to represent the Web application class, and its own benchmark tool to generate the workload. Both, the Web application and the benchmark tool, are realistic and previously used in literature [15, 11]. In these experiments, our approach is able to lead the Web applications according to required resource variations. It allows to control a sequence of resource variations deterministically, by allocating or releasing virtual machines (the resource we considered).

This approach is the first step in direction to test Web applications through elasticity. In a future work, we will address the aspects of the test, such as test case, test strategy and test oracle.

The paper is organized as follows. In the next section, we present the major aspects of elastic Web applications. Section 3 introduces our approach. The experiments and their results are described in Section 4. Section 5 discusses related work. Finally, Section 6 concludes.

2. BACKGROUND

2.1 Cloud Computing Elasticity Behavior

Different authors [1, 2, 7, 3] have a common definition for *cloud computing elasticity*: it is the ability of a cloud infrastructure modifying its resource configuration as quickly as possible, according to application demand.

Figure 1 represents the typical behavior of elastic cloud computing applications. In this figure, the *resource demand* (continuous line) varies over time, at first increasing from 0 to 1.5 (demanding 50% more resources than the current allocated resources) and then decreasing to 0.

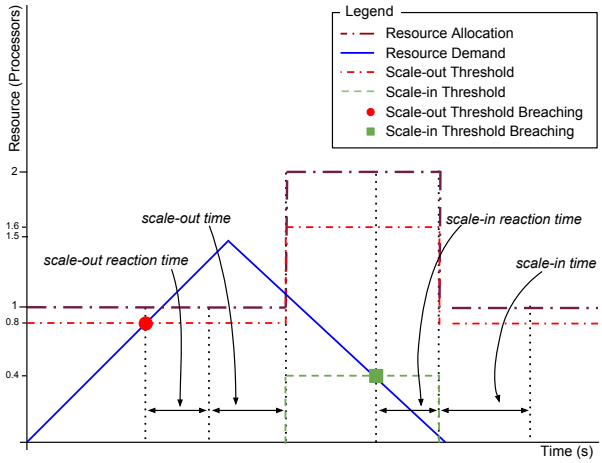

Figure 1: Representation of cloud computing elasticity.

When the resource demand exceeds the *scale-out threshold*, and remains higher during the *scale-out reaction time*, the cloud elasticity mechanisms assign a new resource. The new resource is available after a *scale-out time*, the time the cloud infrastructure spends to allocate the new resource. Once the resource is available, the threshold values are updated accordingly.

When the resource demand decreases, breaches the *scale-in threshold*, and remains lower during the *scale-in reaction time*, the cloud elasticity mechanisms release a resource. As soon as the scale-in begins, the threshold values are updated and the resource is no more available. Nonetheless, the infrastructure needs a *scale-in time* to release the resource.

[1]https://aws.amazon.com
[2]http://jmob.ow2.org/rubbos.html

2.2 Cloud Computing Elasticity States

Fluctuations in workload pressures the Web application. When this application is deployed on a cloud infrastructure, those fluctuations lead to resource variation (elasticity). We classify the resource variation according to resource status, which we call *elasticity states*. Figure 2 illustrates the elasticity states that Web applications running on a cloud infrastructure are exposed.

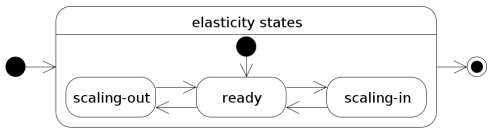

Figure 2: Elasticity states.

At the beginning the application is exposed to the *ready* state, when the resource configuration is steady. Then, if the application is exposed for a certain time (*scale-out reaction time*) to a pressure that breaches the scale-out threshold, the cloud elasticity mechanisms start adding a new resource. At this point, the application is exposed to the *scaling-out* state: period while the resource is added.

After a *scaling-out*, the application returns to the *ready* state. When the application is at the *scaling-out* state, if it is exposed for a certain time (*scale-in reaction time*) to a pressure that breaches the scale-in threshold, the cloud elasticity mechanisms start releasing some resource. This puts the application in the *scaling-in* state: period while the resource is released. After that, the application returns to *ready* state again.

2.3 Workload Variation

In the literature, several work related to Web application benchmark [14, 13, 10, 8] conduct experiments by increasing the workload (intensity, number of clients, etc.) *gradually* over time. In this section, we discuss the effects on requesting elasticity states using this kind of workload fluctuation.

We consider a gradual variation where the workload increases in a rate that requests elasticity changes as soon as possible. At this rate, which we call *hurried rate*, a new scale-out threshold is breached immediately after the previous scaling-out state is completed.

Figure 3 (a) illustrates a *hurried rate*. The workload increases the resource demand in a rate determined by the gradient formula ($m = y/x$). For this calculation, the y is represented by the workload intensity that leads the resource usage to a level that breaches the threshold, while the x is represented by the total scaling time (reaction time + scaling time). In this figure, we also can see that at this rate, the resource is often overused. To avoid the resource over usage, we should increase the workload in a lower rate.

Figure 3 (b) illustrates a *low rate*. This rate is also determined by the gradient formula, though we represent the y by the difference between the total of resource and the resource usage that breaches the threshold. With this rate, the resource is not over used, but the resource variation occurs after a very long time, and moreover, the next one will be very far away. Therefore, the low rate prevents to test sequence of elasticity states, which could last many hours, or even days.

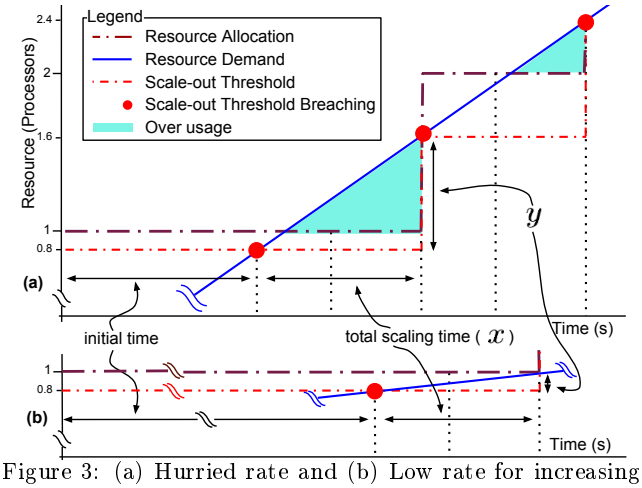

Figure 3: (a) Hurried rate and (b) Low rate for increasing the resource demand gradually.

3. OUR APPROACH

In this section, we describe our approach to lead Web applications through elasticity states. In our approach we request the elasticity states by varying the workload in stages, instead of gradually, attempting to solve the problems discussed in the Section 2.3. We variate the workload in intensity, keeping the same workload pattern during all the leading process.

As workload pattern, we consider the set of requests sent to the Web application, which we represent in the Table 1. In this table, we consider that, except the requests $R2$ and $R3$, all the other requests are executed in parallel. In this case, the requests $R2$ and $R3$ are executed in sequence ($R2$, then $R3$), since the request $R2$ is set as predecessor of $R3$. However, this is only an illustrative case, and other workload patterns may differ from this one. Considering the workload pattern from Table 1, the variation of the workload consists in multiplying the amount of each request by the desired intensity. In this case, all the requests remain the same, only the concurrency is varied.

ID	Request	Amount	Predecessor
$R1$	*GET /index.html HTTP/1.1*	10	–
$R2$	*GET /news.php HTTP/1.1*	5	–
$R3$	*GET /news.php?id=1 HTTP/1.1*	4	$R2$
$R4$	*POST /registration.html HTTP/1.1*	2	–

Table 1: Example of workload pattern.

Figure 4 illustrates our approach to variate the workload intensity, and as a consequence, to request the elasticity states.

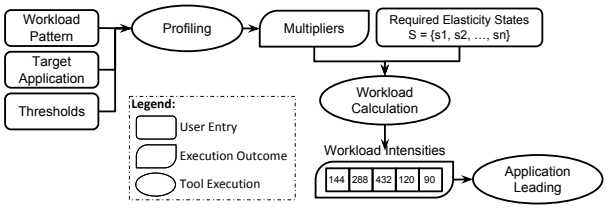

Figure 4: Approach workflow.

At the beginning of our approach, the user sets the following required parameters: workload pattern, target application, thresholds, and required elasticity states. The *workload pattern* describes the set of requests sent to the Web application. The *target application* corresponds to the root Uniform Resource Identifier (URL) of the Web application. The *thresholds* parameter expresses the values of the scale-in and scale-out thresholds set in the cloud computing infrastructure. Finally, the *required elasticity states* expresses the sequence of elasticity states through which the Web application must be led.

After the user set all the required parameters, our approach executes automatically. This execution is divided into three phases: *profiling*, *workload calculation*, and *application leading*.

3.1 Profiling

To lead the target Web application through the required elasticity states, we expose it to the workload intensities that lead the resource usage to levels that breach the thresholds (scale-in, and scale-out). To calculate these intensities, in the profiling phase, we discover which are the workload intensities necessary to breach the thresholds when the resource amount is equal to one. We call these workload intensities as *multipliers*, and use the symbols I_{so} and I_{si} to denote the scale-out and scale-in multipliers, respectively. Then, based on the multipliers, we are able to calculate the intensities necessary to breach the thresholds when the amount of resource is greater than one (Section 3.2).

To discover the multipliers, we expose the Web application to a workload intensity that increases gradually in a *hurried rate* (see Section 2.3) until the scale-out threshold is breached. We use this rate, since it is fast, and does not stress the application until the scale-out threshold is breached. Since the scale-in threshold is lower than the scale-out threshold, going up to the scale-out threshold, allows to discover both multipliers (I_{si}, and I_{so}).

Table 2 shows an example of multipliers. In this example, we consider that the scale-out threshold corresponds to 60% of CPU usage, while the scale-in threshold corresponds to 20% of CPU usage. In this case, we also consider that the discovered scale-out multiplier is equal to 144, while the scale-in multiplier is equal to 30.

Threshold	Threshold Value	Multiplier	Resource Amount
scale-out	$60\%\ of\ CPU\ usage$	$I_{so} = 144$	1
scale-in	$20\%\ of\ CPU\ usage$	$I_{si} = 30$	1

Table 2: Example of multipliers.

3.2 Workload Calculation

In the workload calculation phase, we calculate the workload intensities necessary to lead the Web application through the required elasticity states. For this, each calculated workload intensity should breach the threshold related to an elasticity state.

We base the workload intensities calculation on the multiplier values, and on the results of Lloyd and Smith work [10]. This work shows that multiple web servers (a typical deployment of the Web applications on cloud) scale near-linearly ($> 99\%$). Therefore, we assume that the workload intensity necessary to breach the threshold when the amount of resource is greater than one, is equal to the current amount of resource multiplied by the corresponding multiplier, i. e., I_{so} for scale-out, or I_{si} for scale-in.

Table 3 exemplifies the calculation of the workload intensities for a sequence of elasticity states. In this table, the required states correspond to a sequence of three scale-outs and two scale-ins, where after each elasticity state the amount of resource is increased or decreased accordingly.

Elasticity State	Formula	Workload Intensity	Resource Amount
scale-out	$1I_{so}$	144	1
scale-out	$2I_{so}$	288	2
scale-out	$3I_{so}$	432	3
scale-in	$4I_{si}$	120	4
scale-in	$3I_{si}$	90	3

Table 3: Example of workload intensities calculation.

3.3 Application Leading

During the application leading phase, we lead the Web application through the desired elasticity states. This consists in exposing the Web application to each one of the workload intensity calculated in the previous phase until the elasticity state related to the intensity is completed. This phase can be better understood in the description of the Figure 6 (next section).

3.4 Implementation Architecture

In this section, we explain the implementation of our approach. Figure 5 illustrates the components of the implementation architecture, which is composed by a *coordinator*, and several *generators*.

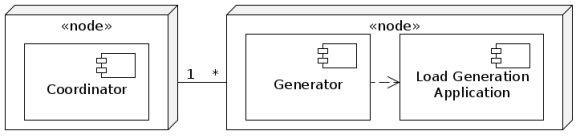

Figure 5: Implementation architecture of our approach.

The *coordinator* has several roles: front-end, monitoring, calculation of workload intensity variations and synchronization of load tasks. Each *generator* runs and controls an instance of a *load generation application*, leading it to perform the load tasks received from the coordinator. Using multiple generators reproduces a more realistic workload than using a single instance, allows to generate high workload intensities, and may prevent false-positive attacks.

Figure 6 shows the sequence diagram that represents the implementation of the application leading phase. In this diagram, for each calculated workload intensity, the *coordinator* splits the workload generation task among the multiple *generators*. Then, each *generator* generates its portion of the workload, and when the *coordinator* identifies that the required elasticity state is performed, it sends the next workload intensity to the *generators*. This process is repeated until all of the workload intensities are generated. To identify the elasticity states, the *coordinator* monitors the resource status on the *cloud frontend*.

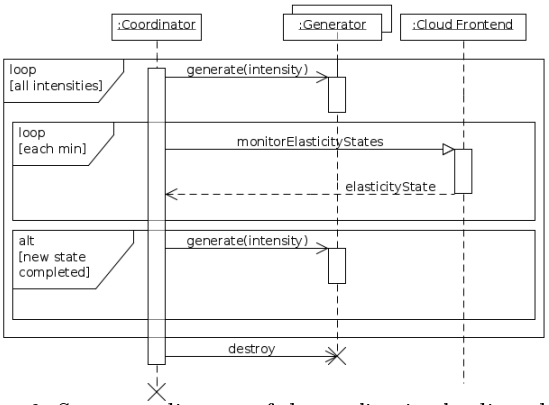

Figure 6: Sequence diagram of the application leading phase.

In the current version, our tool is implemented in Java. All the parameters are set using a property file. The synchronization tasks are performed by *remote method invocation* (RMI). The generators run any load generation application that has an API, or supports command line execution. If an *application programming interface* (API) is available, the monitoring tasks are performed using it. Otherwise, a virtual machine used to host the web application is accessed remotely, and the *sysstat*[3] tool is used for monitoring. The remote access is also used if frequent monitoring (e. g., every second) is necessary, since this is usually not allowed by the cloud providers APIs. In the case where remote access is not allowed, our tool can also monitor the cloud infrastructures by reading the resource status on cloud provider status pages.

4. EXPERIMENTS AND DISCUSSION

In this section, we present the experiments and discuss their results. All the experiments are conducted on Amazon EC2, where we set the scale-out threshold as 60% of CPU usage, and the scale-in threshold as 20% of CPU usage. The workload *multipliers* that breach these thresholds are predicted as: 144 (I_{so}), and 30 (I_{si}), the same as the values of Section 3.1. The *total scale-out time* is equal to 360 seconds, while the *total scale-in time* is equal to 120 seconds.

We represent the Web application with the PHP version of the *RUBBoS*, a realistic web application modeled after an online news forum like Slashdot[4]. The RUBBoS is deployed on n web servers with a centralized database server. For each web server we use a distinct virtual machine with a standard capacity ($m3.medium$). The database server is deployed on a large capacity machine ($m3.large$) at same geographic region of the web servers, which avoids bottlenecks.

We generate the workload using the RUBBoS benchmark tool[5], where the workload intensity is varied in number of clients. For the first two experiments, the RUBBoS benchmark is executed natively, and it is distributed over 10 machines. For the third experiment, we use our tool to control the RUBBoS benchmark. Our tool is deployed with 1 coordinator and 10 generators. In all the experiments, the RUBBoS benchmark is deployed at same cloud provider and geographic region of the Web application, which prevents bandwidth problems. All the machines used to deploy the RUBBoS benchmark are of the large capacity type.

Table 4 describes the configuration of each machine type used in the experiments.

Machine Type	CPU	Memory	Disk
$m3.medium$	1 vCPU (2.4 GHz)	3.7 GB	10 GB
$m3.large$	2 vCPU (2.4 GHz)	7.5 GB	10 GB

Table 4: Machines configuration.

4.1 Gradual Workload Variation in a Hurried Rate

This first experiment is conducted to confirm our assumption about leading Web applications through the required elasticity states by using a workload that varies gradually over time in a *hurried rate*. The hurried rate is calculated according to the gradient formula from Section 2.3. This results in an increasing rate equals to 0.4 (144/360) per second,

[3]http://sebastien.godard.pagesperso-orange.fr
[4]http://slashdot.org/
[5]Configuration of the benchmark: https://goo.gl/nMm0ro.

and a decreasing rate equals to 0.25 (30/120) per second. We first increase the workload intensity from 0 to 1152, and at the end of this increase we keep the workload at the higher intensity for a certain time (*total scale-out time*), then we decrease it from 270 to 0. This results in an execution time bigger than one hour (enough time to gather the results that confirm our assumption).

Increasing the workload intensity up to 1152 gradually in a hurried rate should lead the Web application to 7 scale-outs (8 machines), based on the calculations that we make in the Section 3.2. Also based on the calculation of the Section 3.2, with 8 machines, the workload intensity necessary to request a scale-in is equal to 240.

At the workload increase, the workload start being increased from 0, which lasts a little ($\approx$ *total scale-out time*) until the first scale-out threshold is breached. To reproduce a beginning such as in the workload increase step, we set 270 ($240 + 30 = 270$, where 30 is the scale-in intensity factor) as the value from which the workload starts decreasing. Thus, the workload is decreased during a certain time ($\approx$ *total scale-in time*) before breaching the first scale-in threshold.

Figure 7 illustrates the behavior of the Web application, and the cloud infrastructure during this experiment.

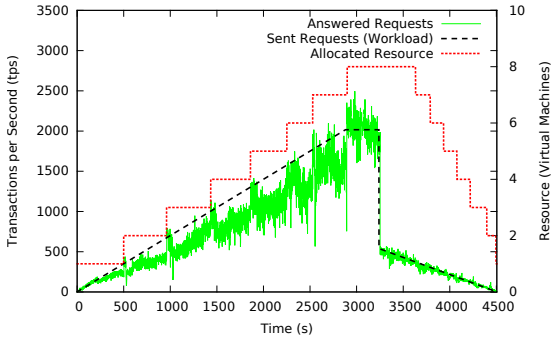

Figure 7: Gradual workload variation in a *hurried rate*.

In this figure, we can see that happen seven scale-outs, and seven scale-ins, according to the expected behavior. However, if we compare the *Sent Requests (Workload)* and the *Answered Requests* lines, we see that most of the time that the workload increases, the amount of answered requests is lower than the amount of sent requests. This behavior represents periods of stress, which occur due to the over usage of resource (see Section 2.3). Therefore, this experiment confirms our assumption that requesting elasticity states by varying the workload gradually in a hurried rate stresses the application.

4.2 Gradual Workload Variation in a Low Rate

This second experiment intends to confirm our assumption about leading Web applications though the required elasticity states by using a workload that varies gradually over time in a *low rate*. Since this experiment would take too long ($\approx$ 9 hours) to be entirely executed, we only execute it during two scale-outs, which is enough to confirm our assumption. With these two scale-outs, we can measure the *total scale-out time*, and see whether the Web application is stressed (wherein the *hurried rate* stresses the application even before the first scale-out). We consider the *hurried rate* for the scale-ins, since with this rate (in the previous experiment) the scale-ins are requested quickly, and without stressing the Web application.

Figure 8 illustrates the results of this experiment execution.

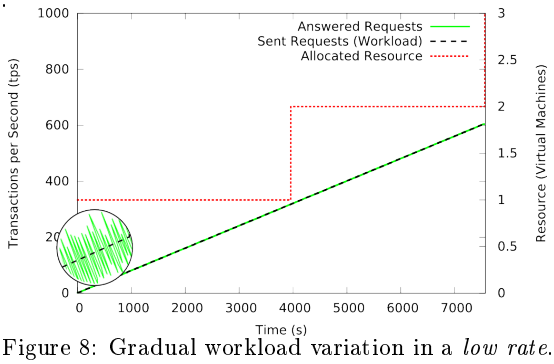

Figure 8: Gradual workload variation in a *low rate*.

In this figure, the lines that represent the *Sent Requests (Workload)* and the *Answered Requests* are very close. Since the execution time of this experiment is long, and the variation of the answered requests is low, it is not possible to see the difference between the both lines without enlarging the figure. Therefore, we use a enlarged part of the figure to show more precisely the difference between the sent requests and the answered requests. In this enlarged part, we can see the answered requests line variate more than the sent requests line. This variations happen since the RUBBoS benchmark (used to generate the workload) uses a *thinking time*, which delays some requests, mimicking the real users behavior. However, this is not a behavior related to the stress of the Web application. The execution time of this experiment is longer than in the previous experiment, even that we request only two scale-outs, instead of seven scale-outs, and seven scale-ins. This confirms our second assumption about gradual workload variation, where requesting elasticity states by varying the workload gradually in a low rate takes too long.

4.3 Our Tool Validation

This last experiment is conducted to validate the ability of our approach leading Web applications through the required elasticity states. For that, we use the same elasticity states of the first experiment, i. e., 7 scale-outs, followed by 7 scale-ins. We variate the workload intensity step-by-step, according to the calculated workload intensities (see Section 3.2):

- *scale-out:* 144, 288, 432, 576, 720, 864, 1008, 1152;
- *scale-in:* 240, 210, 180, 150, 120, 90, 60;

Figure 9 illustrates the results of this experiment execution.

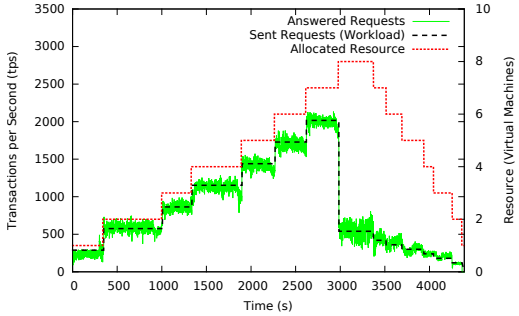

Figure 9: Step-by-step (controlled) workload variation.

In this figure, the relation between the sent requests (*Sent Requests (Workload)*) and the answered requests (*Answered*

Requests) shows that the Web application is not stressed at any time. The total of allocated machines increases from 1 to 8, and then decreases from 8 to 1, according to expected. Our experiment executes within the same time than when the workload increases in a *hurried rate*. Therefore, in addition to lead the Web application through the required elasticity states, our approach also solves the issues about gradual workload variation, since it requests the elasticity states in a minimal time, without stressing the application.

5. RELATED WORK

Gambi et al. have two work that address elasticity testing [6, 4]. In the first work, the authors predict elasticity state transition based on workload variations, and test whether cloud infrastructures react accordingly. The second work presents a tool for automatic testing cloud-based elastic systems, however this tool does not take into account elasticity states. Our approach is similar to their first work in two points: both trigger resource variations without stress, and consider elasticity states. However, the authors classify the elasticity states according to amount of resource allocated, while we also consider as elasticity state the periods where the resource is being allocated. We also control the elasticity state, while they do not.

Malkowski et al. [12] focus on controlling the elasticity of n-tier applications, such as Web applications. This work is similar to ours in two points: it addresses n-tier applications, and it bases its models on previous application runs. Although, it addresses the cloud infrastructure side, while our approach interacts directly with application (opposite side).

Other work are related to web performance measurement [14, 13, 10, 8]. The Web performance is also addressed by commercial tools, such as Apache JMeter[6] and Blazemeter[7]. However, none of them addresses the elasticity.

6. CONCLUSION

In this paper we propose an approach that automates and controls the workload generation in order to lead Web applications through elasticity states. Our approach gathers the unknown elasticity variables by experimentation, then calculates the workload variations, and leads the web application through a list of required elasticity states automatically.

The results of the first and second experiments (Sections 4.1 and 4.2) indicates that the gradual variation of the workload is not a good way to request the elasticity states. In these experiments we see that when the elasticity states are requested in a *hurried rate*, the Web application is stressed. In addition, to not stress the Web application we experiment the application with a *low rate*, which extends the time of requesting the elasticity states. The third experiment (Section 4.3) shows that our approach generates the workload variations that lead the web applications through the required elasticity states. In this experiment, the web application is led without stress, and in a time close to the Experiment 1 (fastest). Therefore, we can say that our approach is able to lead web applications through the required elasticity states, allowing a reduced execution time.

This paper is the first step towards an approach to test Web applications through elasticity. At the moment, we lead web applications through required elasticity states. Although, we are also interested on aspects related to the test, such as test case and test oracle. As a future work, we intend to propose an strategy to test Web applications through elasticity.

[6]http://jmeter.apache.org/

[7]https://blazemeter.com/

Acknowledgments

This work is supported by CAPES Fundation (Science Without Borders process 9070–13–3), Ministry of Education of Brazil.

7. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore. Database scalability, elasticity, and autonomy in the cloud. *Proceedings of DASFAA'11*, pages 2–15, Apr. 2011.

[2] L. Badger, T. Grance, R. Patt-Comer, and J. Voas. *Draft Cloud Computing Synopsis and Recommendations*. Nist Special Publication 800-146, 2011.

[3] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić. Towards the Formalization of Properties of Cloud-based Elastic Systems. In *Proceedings of PESOS 2014*, pages 38–47, New York, NY, USA, 2014. ACM.

[4] A. Gambi, W. Hummer, and S. Dustdar. Automated testing of cloud-based elastic systems with AUToCLES. In *Conference ASE 2013*, pages 714–717. IEEE, Nov. 2013.

[5] A. Gambi, W. Hummer, and S. Dustdar. Testing elastic systems with surrogate models. In *Workshop CMSBSE'13*, pages 8–11. IEEE, May 2013.

[6] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, 2013.

[7] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. *ICAC*, pages 23–27, 2013.

[8] W. Iqbal, M. Dailey, and D. Carrera. SLA-Driven Dynamic Resource Management for Multi-tier Web Applications in a Cloud. In *Proceedings of CCGrid'10*, pages 832–837, May 2010.

[9] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of ICPE'12*, page 85, New York, New York, USA, Apr. 2012. ACM Press.

[10] C. U. S. Lloyd G. Williams. Web Application Scalability: A Model-Based Approach. pages 215–226, 2004.

[11] S. Malkowski, M. Hedwig, D. Jayasinghe, J. Park, Y. Kanemasa, and C. Pu. A new perspective on experimental analysis of N-tier systems: Evaluating database scalability, multi-bottlenecks, and economical operation. In *Proceedings of CollaborateCom'09*, pages 1–10, Nov. 2009.

[12] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of ICAC '11*, pages 131–140, New York, USA, June 2011. ACM Press.

[13] D. Menasce. Load testing of Web sites. *IEEE Internet Computing*, 6(4):70–74, July 2002.

[14] D. Mosberger and T. Jin. httperf - A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, Dec. 1998.

[15] Q. Wang, Y. Kanemasa, J. Li, C.-A. Lai, C.-A. Cho, Y. Nomura, and C. Pu. Lightning in the Cloud: A Study of Transient Bottlenecks on n-Tier Web Application Performance. 2014.